CS 60050
Machine Learning

Neural Networks

Some slides taken from course materials of Abu Mostafa

Gradient Descent – as we studied it

- GD minimizes

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \underbrace{\mathrm{e}\left(h(\mathbf{x}_n), y_n\right)}$$

- Δparameter = - learning rate * gradient
- Gradient computed based on all training examples (x_n, y_n): “Batch” GD
- Epoch: using all training examples once

Stochastic Gradient Descent (SGD)

- Pick one (x_n, y_n) at a time, apply GD to $e(\ h(x_n), y_n\ )$
- When done over many training examples, many times, average direction of descent will be the same as the "ideal" direction
- Benefits
  - Cheaper computation
  - Randomization helps escape trivial local minima
  - Like batch GD, cannot guarantee reaching global minima for non-convex error functions (most error functions, especially in neural networks, will be non-convex)

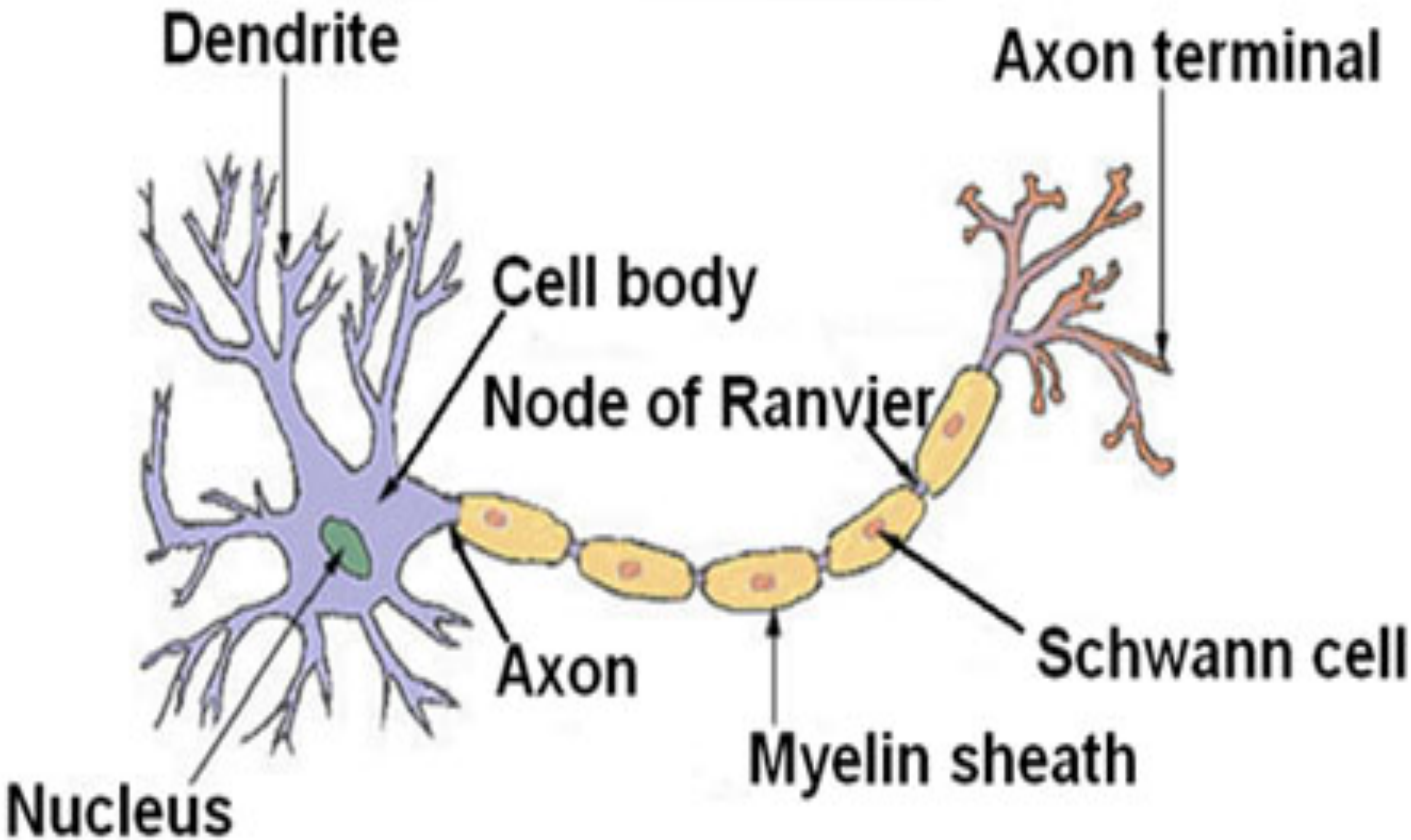
Limitations of linear models

- Linear models not sufficient for regression / classification of complex functions

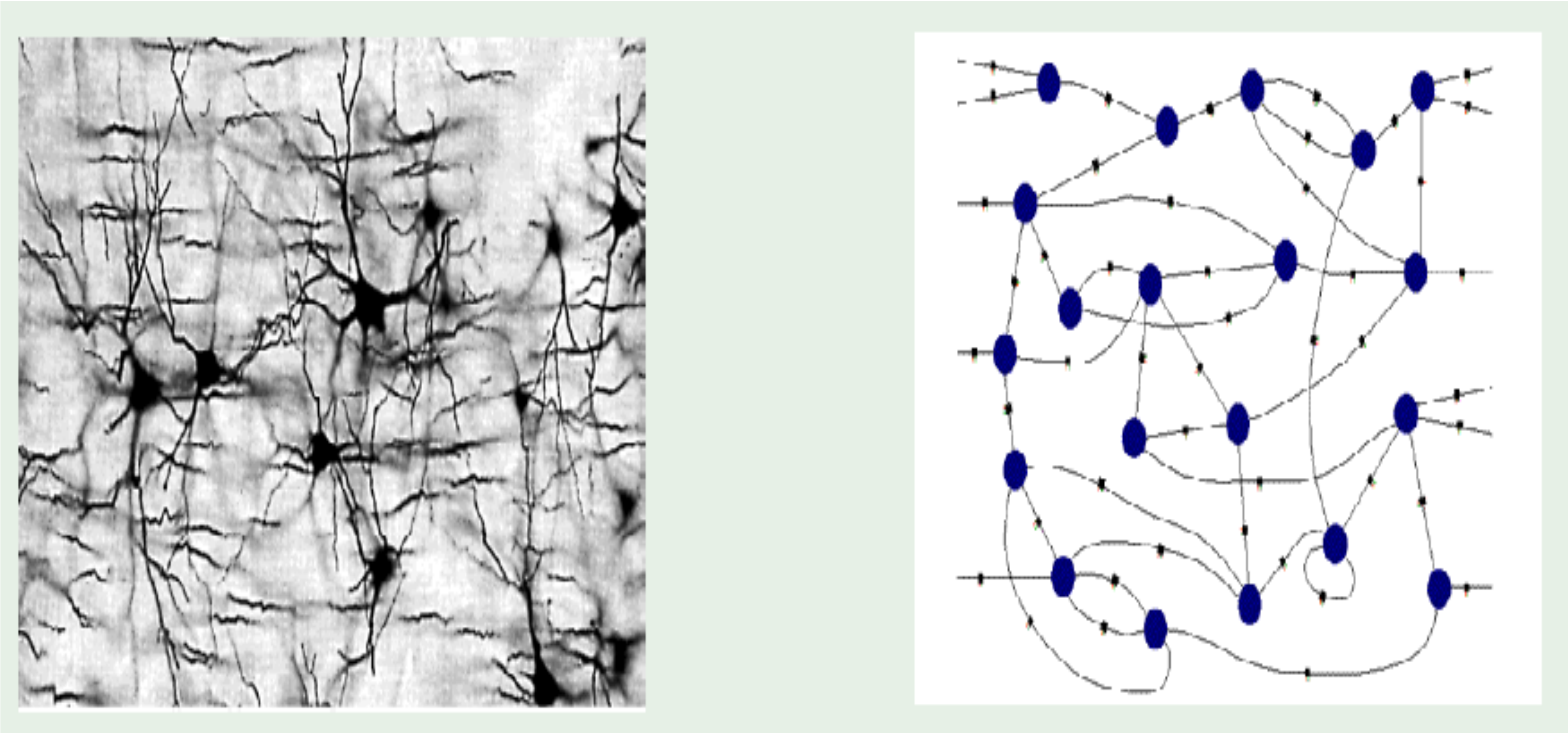
- Non-linear combinations can be used, but not feasible as the number of features increases beyond few hundred (e.g., pixels in an image) – which non-linear combinations to use?

- Need for non-linear models

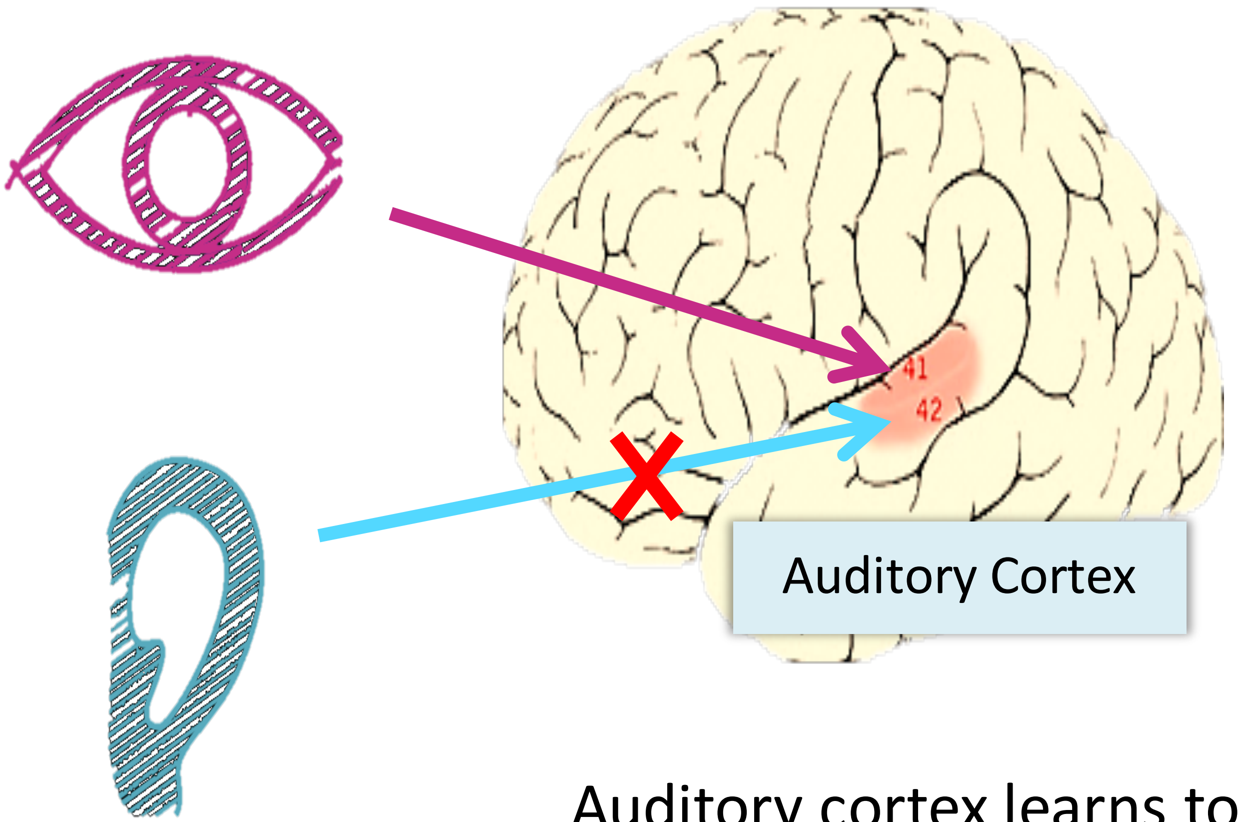
Neural Networks: Algorithms that try to mimic the brain



Idea: To mimic the biological function, first mimic the biological structure



The "one learning algorithm" hypothesis



Auditory cortex learns to see

Neural Networks

- Was very widely used in 80s and early 90s
- Popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
  - Better hardware infrastructure (GPUs)
  - Better algorithms to deal with some problems in earlier implementations

Logical unit: perceptron

- Inputs x_1, x_2, … each take values {-1, +1}
- One input is a constant (called a bias)

- Each input x_i has a weight w_i

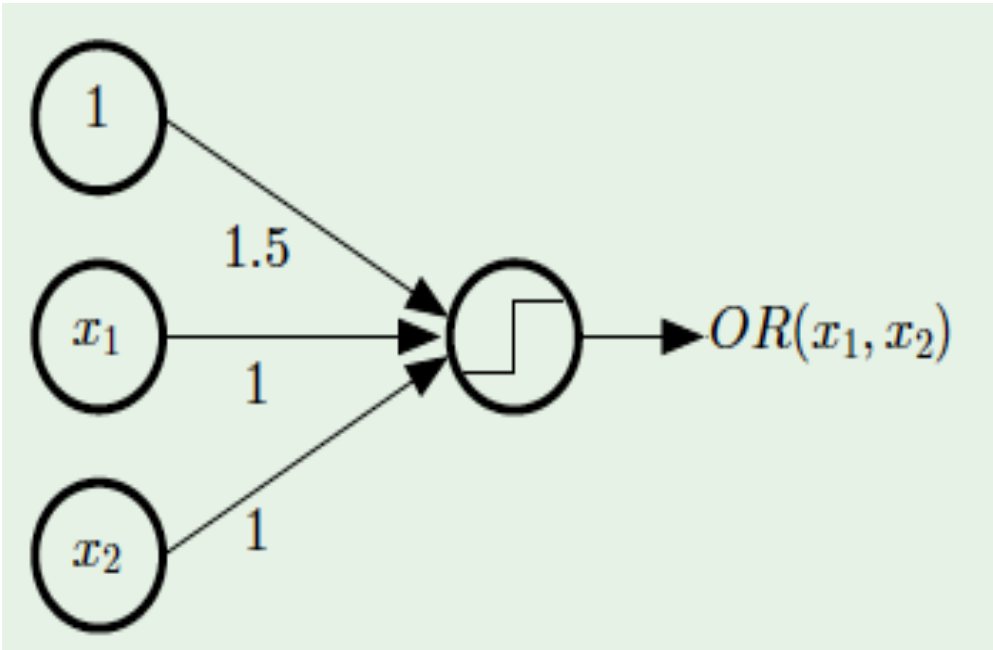
- Output: weighted sum of inputs = $\sum w_i x_i$

- Convention for both inputs and output: negative means logical 0, positive means logical 1

Using perceptron for logical operation (OR)

Inputs x_1, x_2, … each take values {-1, +1}
Output: weighted sum of inputs = $\sum w_i x_i$

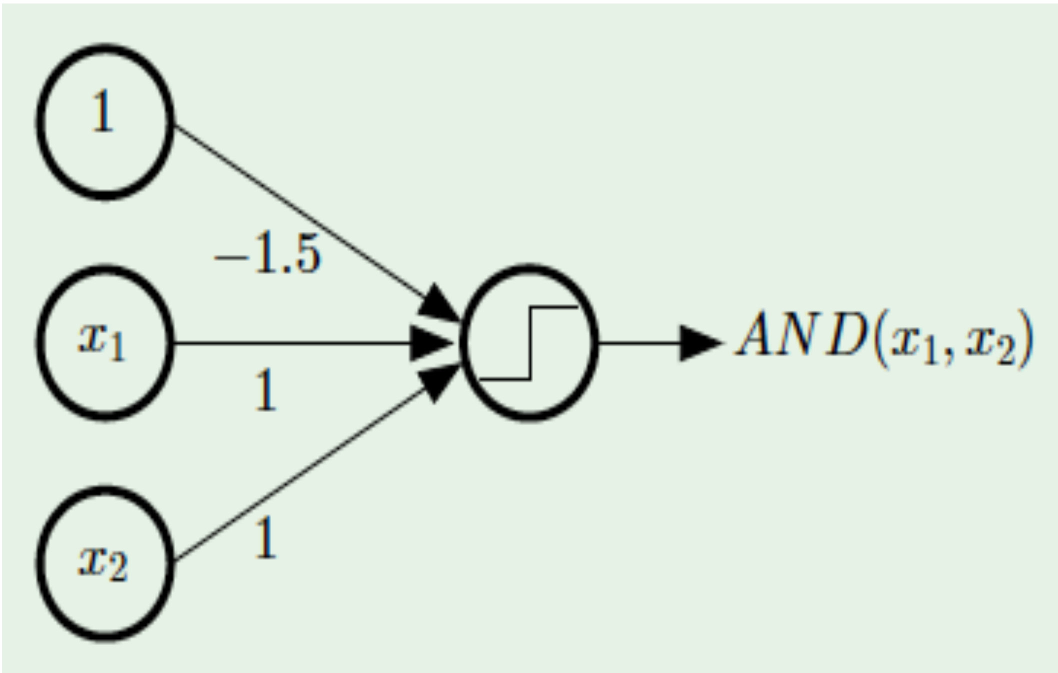
Convention for both inputs and output: negative means logical 0, positive means logical 1



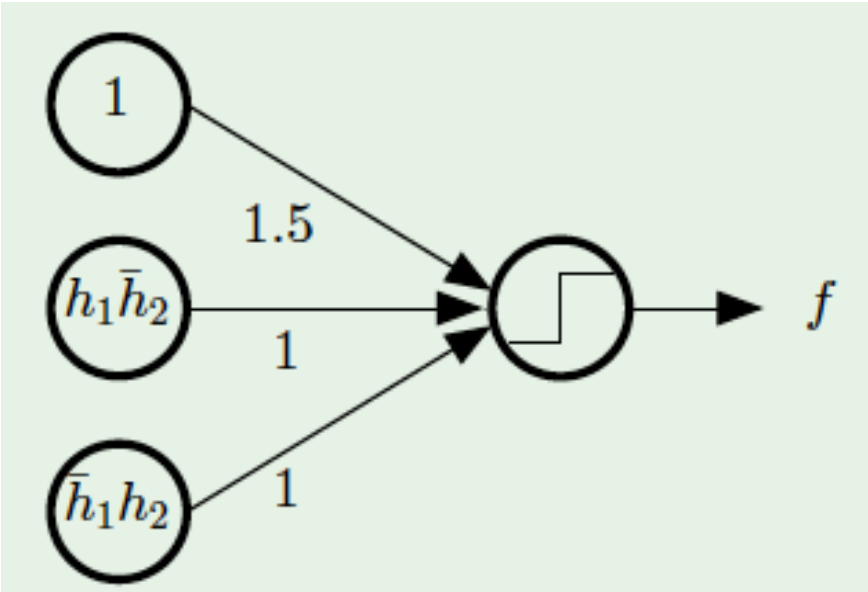
Using perceptron for logical operation (AND)

Inputs x_1, x_2, … each take values {-1, +1}
Output: weighted sum of inputs = $\sum w_i x_i$

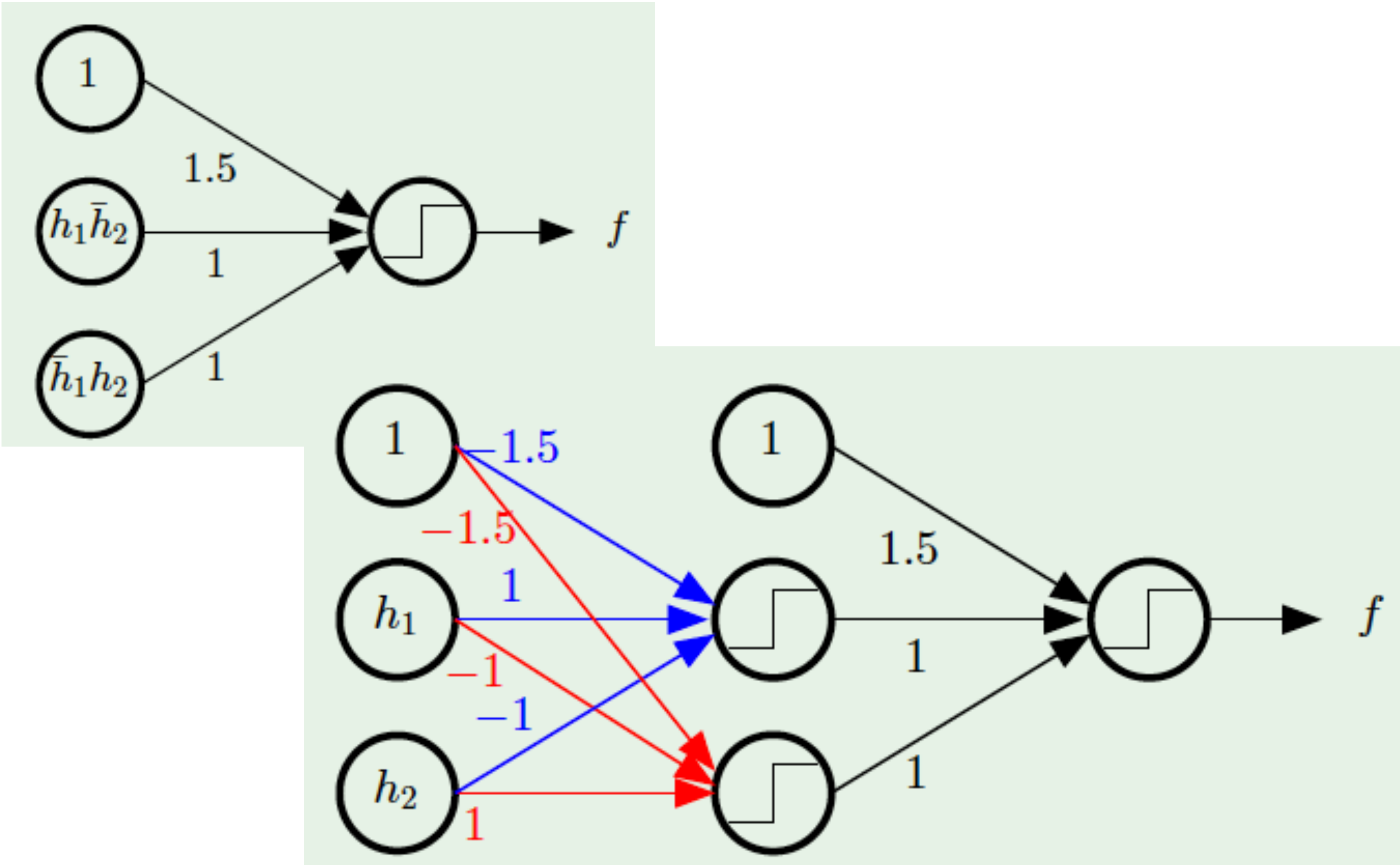
Convention for both inputs and output: negative means logical 0, positive means logical 1



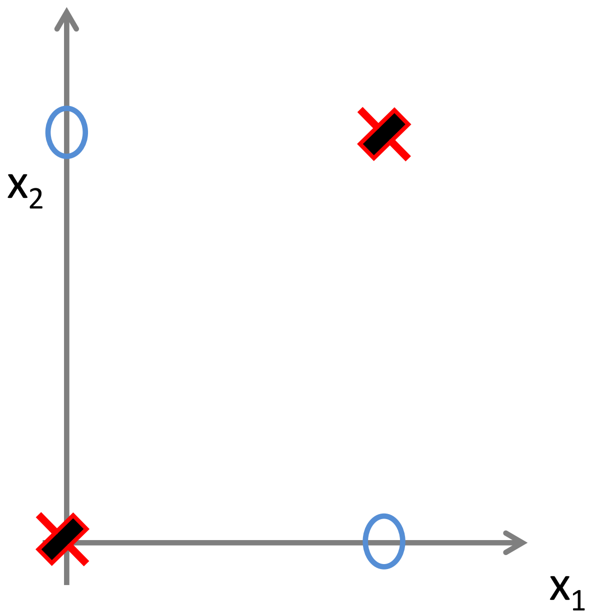
Creating layers of perceptrons to implement more complex functions (XOR)



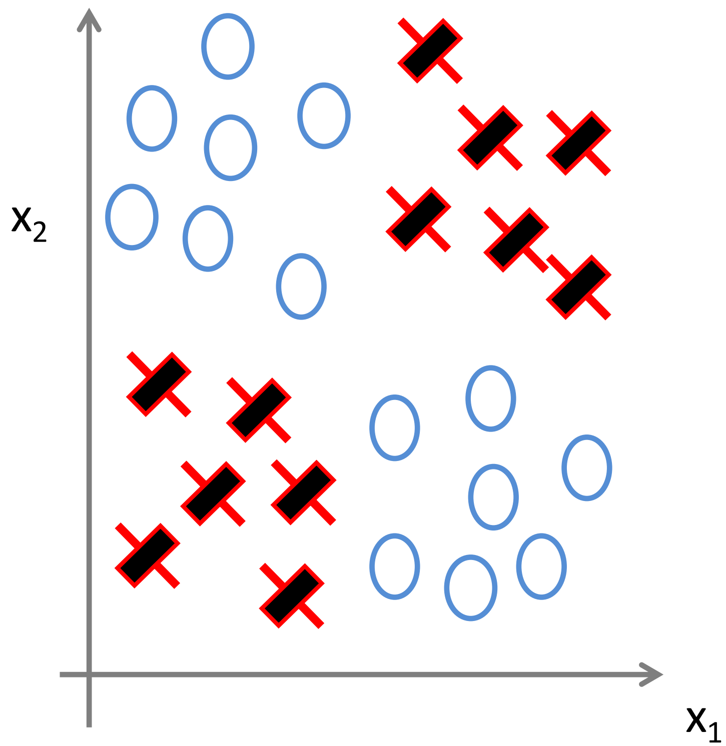
Creating layers of perceptrons to implement more complex functions (XOR)



Non-linear classification using perceptrons

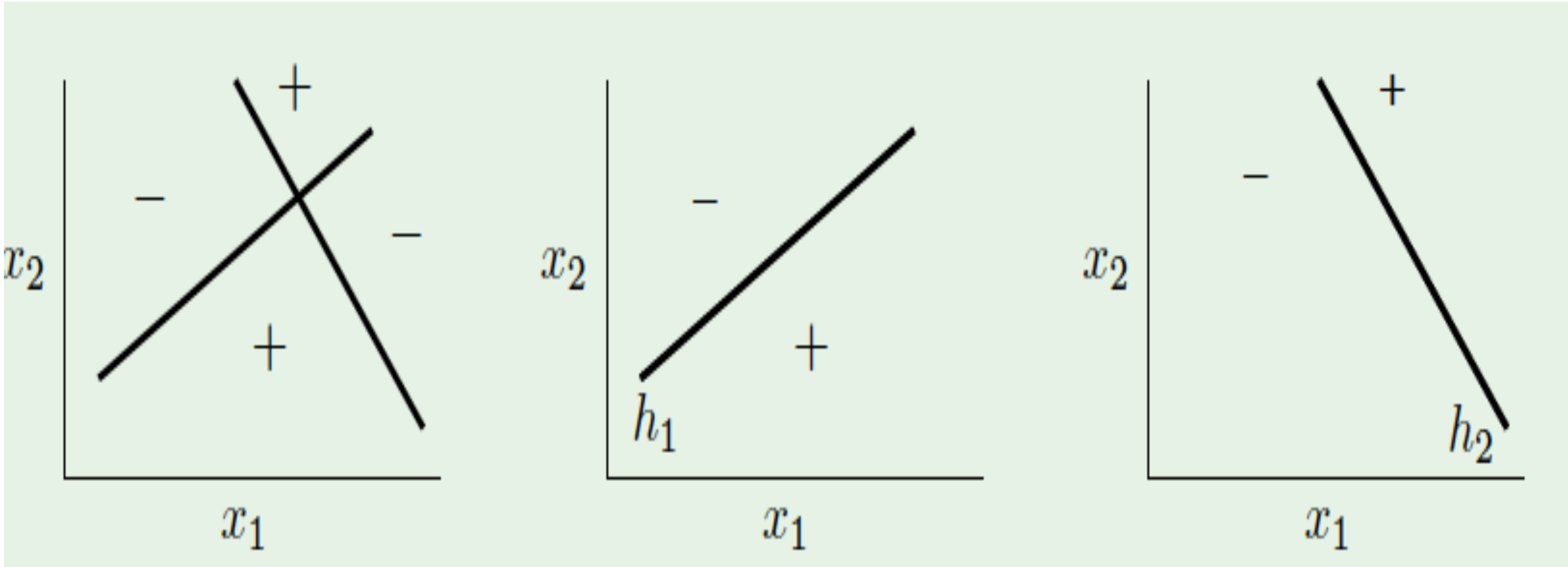


$$y = x_1 \text{ XOR } x_2$$
$$x_1 \text{ XNOR } x_2$$
$$\text{NOT } (x_1 \text{ XOR } x_2)$$

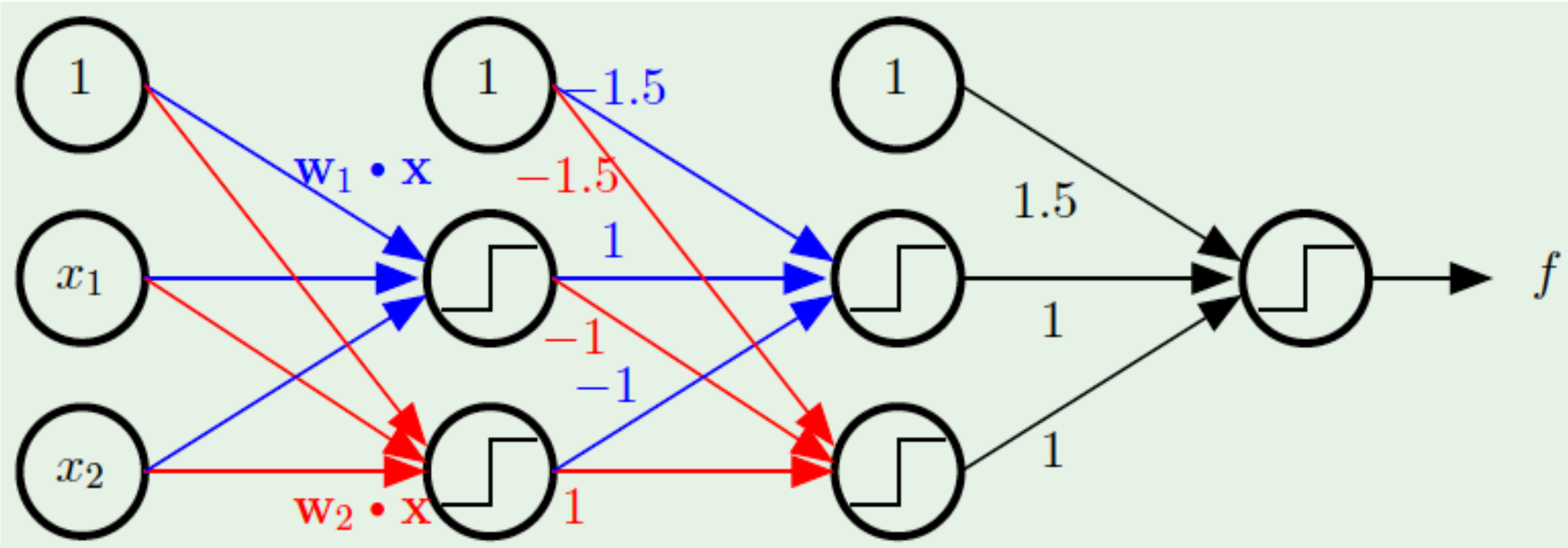


Cannot be separated using a perceptron or any linear classifier model

Need to combine multiple perceptrons

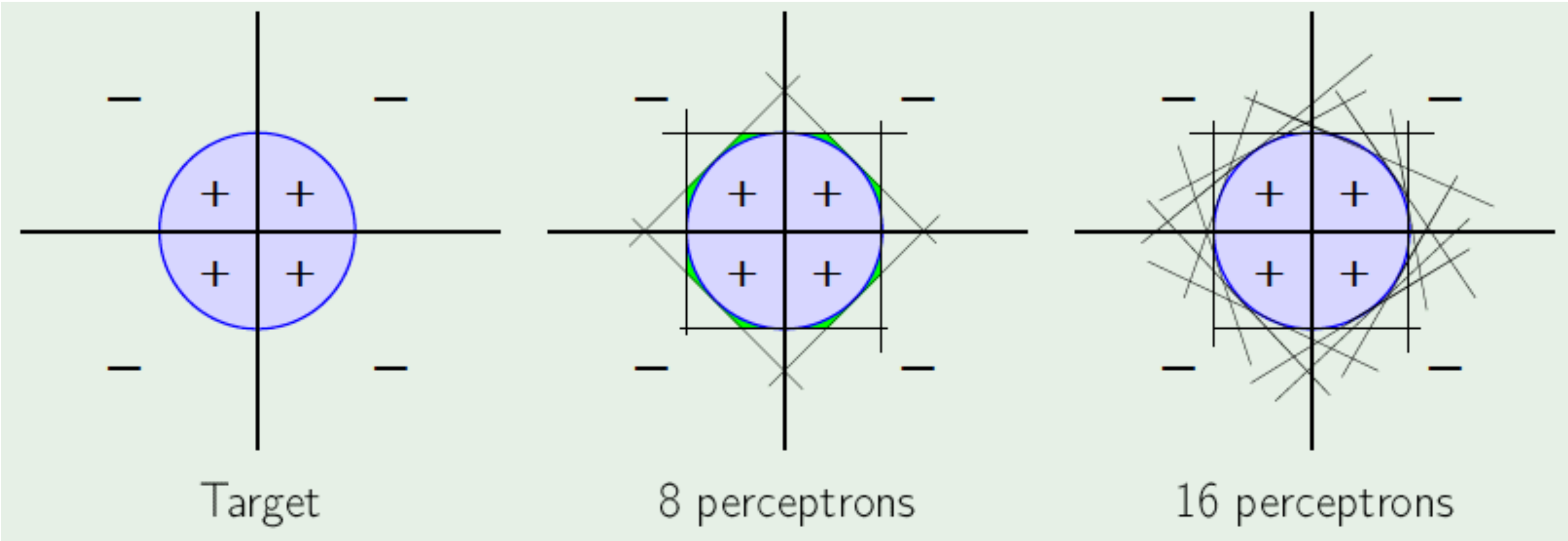


A multi-layer perceptron for the non-linear classification



Suitable weights w_1 and w_2 need to be fixed

A powerful model – can generate complex decision boundaries by combining many linear classifiers



Multilayer perceptrons, suitably combined, can generate almost all functions / decision boundaries

From perceptron to a neuron

- Optimization becomes difficult with many perceptrons
- Desirable: instead of a hard threshold of the perceptron, a smooth function that is efficient to differentiate
- A perceptron with a smooth non-linear function is called a neuron

From perceptron to a neuron

- Desirable: a smooth function that is efficient to differentiate
- Possible functions
  - Range [0, 1]: logistic function
  - Range [-1, 1]: tanh function

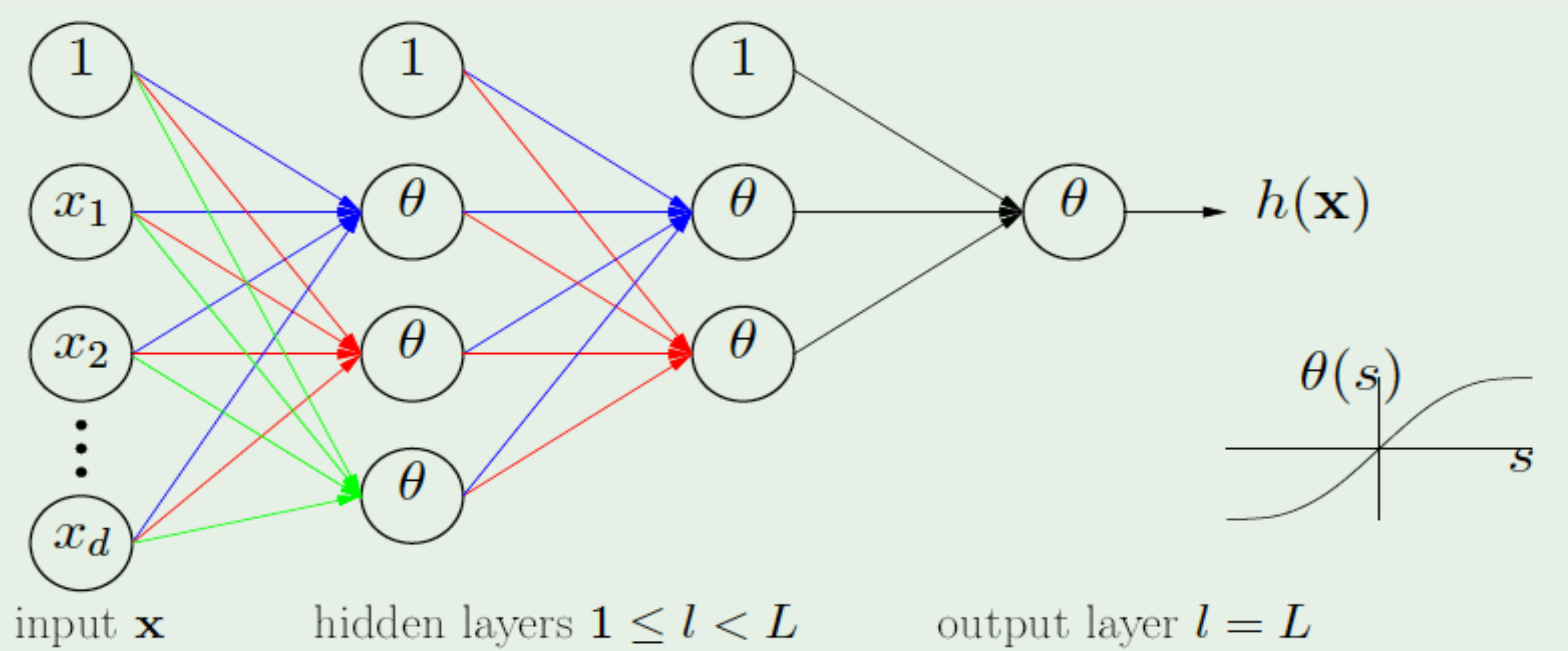
Logistic function

$$\Theta(z) = \frac{1}{1+e^{-z}}$$

tanh function

$$\Theta(s) = \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

A neural network



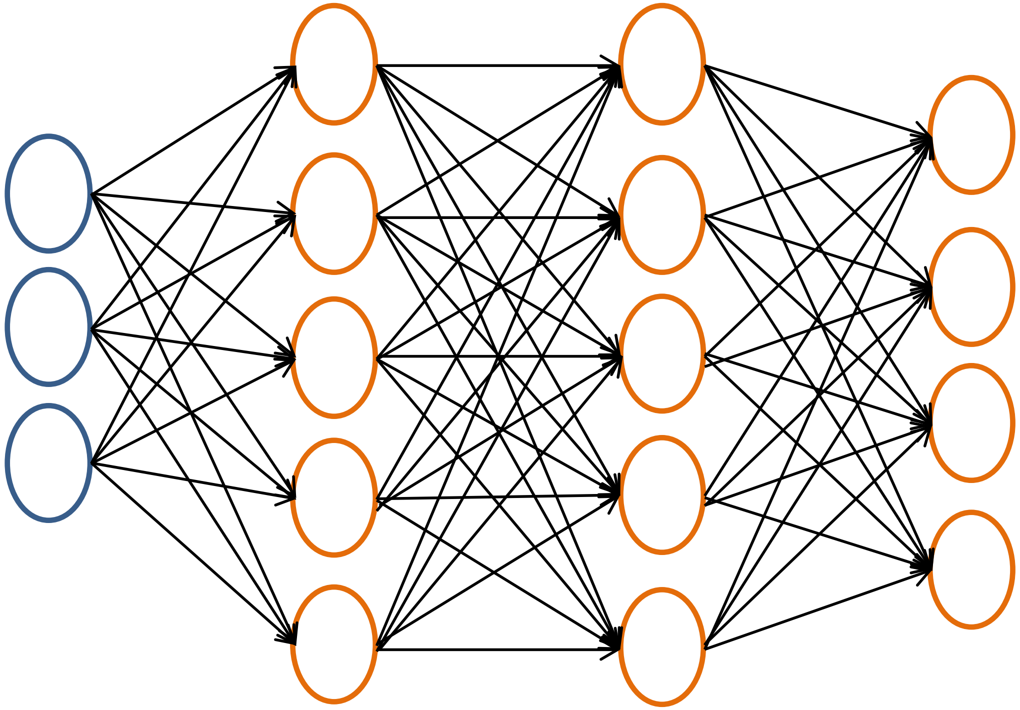
Number of layers: L

Input layer, hidden layer(s), output layer

Number of neurons in layer l: $d^{(l)}$

Number of neurons in input layer = number of features in input = $d^{(0)}$

Different architectures possible for neural network. Example for a four-class classifier



For our discussion:

- We consider a simple regression model with only one neuron in the output layer
- Non-linearity in different neurons can be different. We consider all neurons to implement the same non-linear function

How the network operates

$$w_{ij}^{(l)} \quad \begin{cases} 1 \le l \le L & \text{layers} \\ 0 \le i \le d^{(l-1)} & \text{inputs} \\ 1 \le j \le d^{(l)} & \text{outputs} \end{cases}$$

Weight of the link from i-th neuron in layer (l-1) to the j-th neuron in layer l

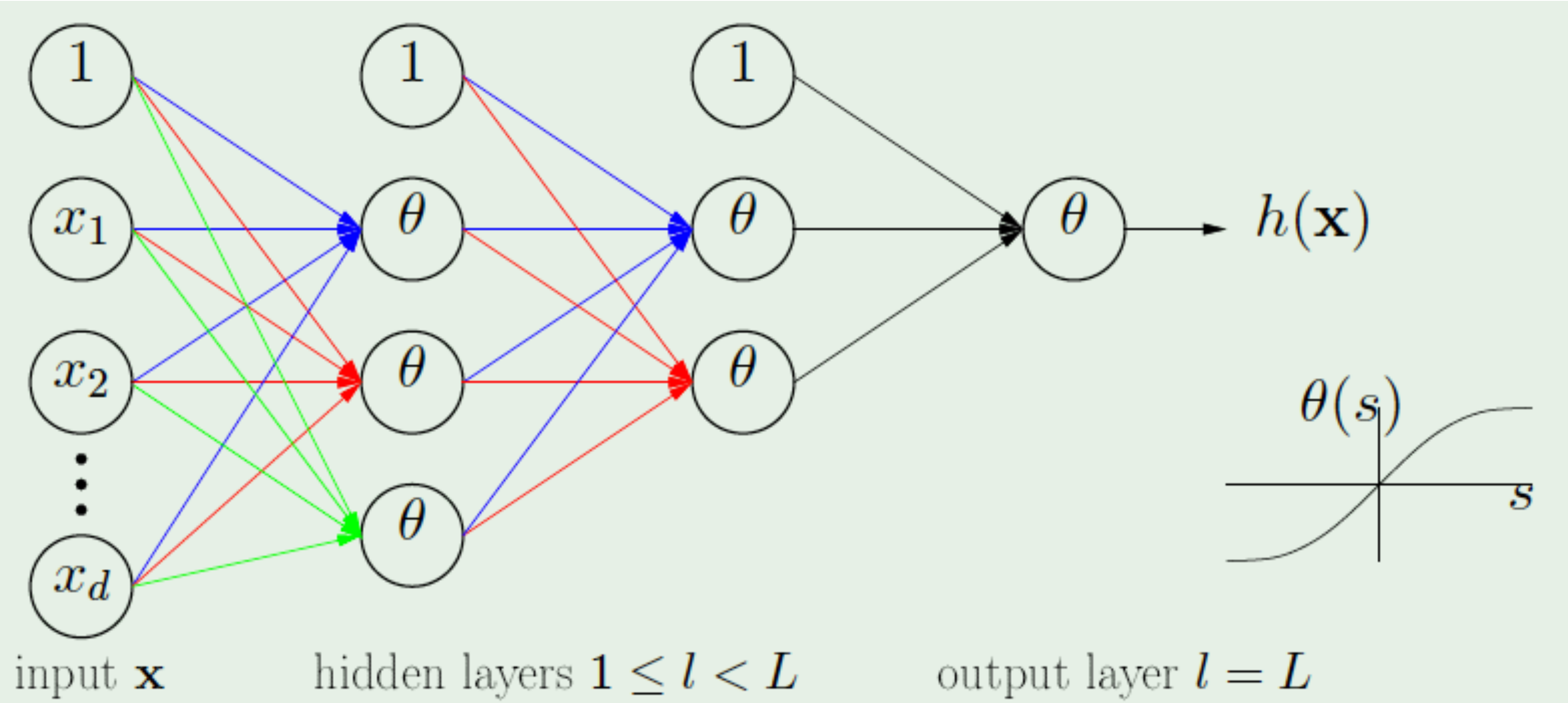
$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} \, x_i^{(l-1)}\right)$$

Output of the j-th neuron in layer l

Input to the j-th neuron in layer l

How the network operates

Apply $\mathbf{x}$ to $x_1^{(0)} \cdots x_{d^{(0)}}^{(0)} \to \to x_1^{(L)} = h(\mathbf{x})$



How to get the weights?

- Till now what we have discussed – if the weights are known, how the neural network operates
- As ML practitioners, our job is to automatically learn the weights from training data
- Learning the weights efficiently: Backpropagation algorithm

Applying SGD

- All weights $w = \{ w_{ij}^{(l)} \}$ determine the hypothesis h(x)
- Error on example (x_n, y_n) is $e(h(x_n), y_n) = e(w)$ which can be squared error or logistic error function
- To implement SGD, we need the gradient

$$\nabla \mathrm{e}(\mathbf{w}): \quad \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ w_{ij}^{(l)}} \quad \text{for all} \quad i, j, l$$

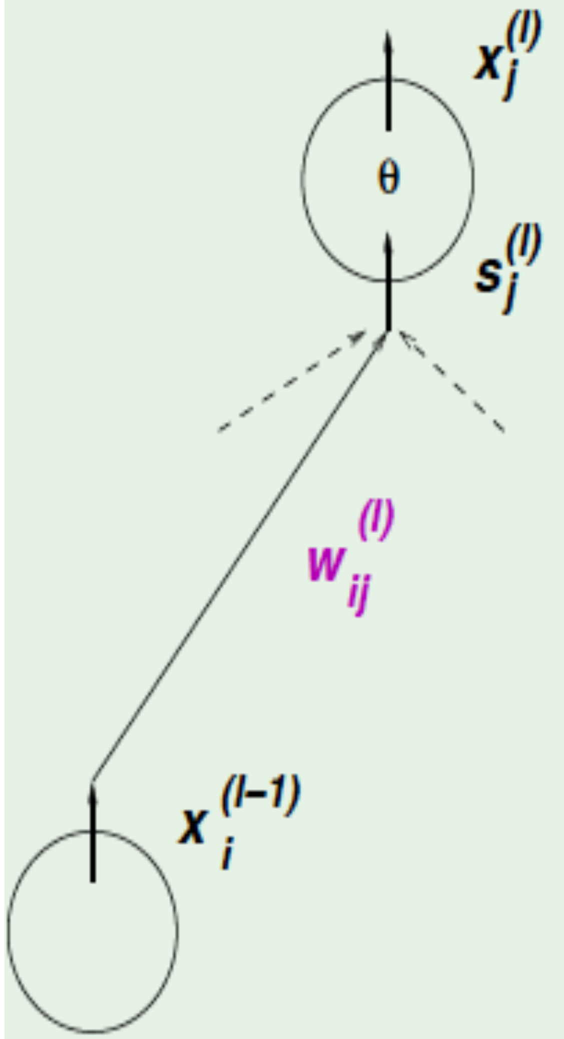
- Can compute the differentials one by one, analytically or numerically, but it will be very inefficient

Computing $\dfrac{\partial \ \mathrm{e}(\mathbf{w})}{\partial \ w_{ij}^{(l)}}$

A trick for efficient computation:

$$\frac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, w_{ij}^{(l)}} = \frac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, s_j^{(l)}} \times \frac{\partial \, s_j^{(l)}}{\partial \, w_{ij}^{(l)}}$$

We have $\dfrac{\partial \, s_j^{(l)}}{\partial \, w_{ij}^{(l)}} = x_i^{(l-1)}$ We only need: $\dfrac{\partial \, \mathrm{e}(\mathbf{w})}{\partial \, s_j^{(l)}} = \delta_j^{(l)}$

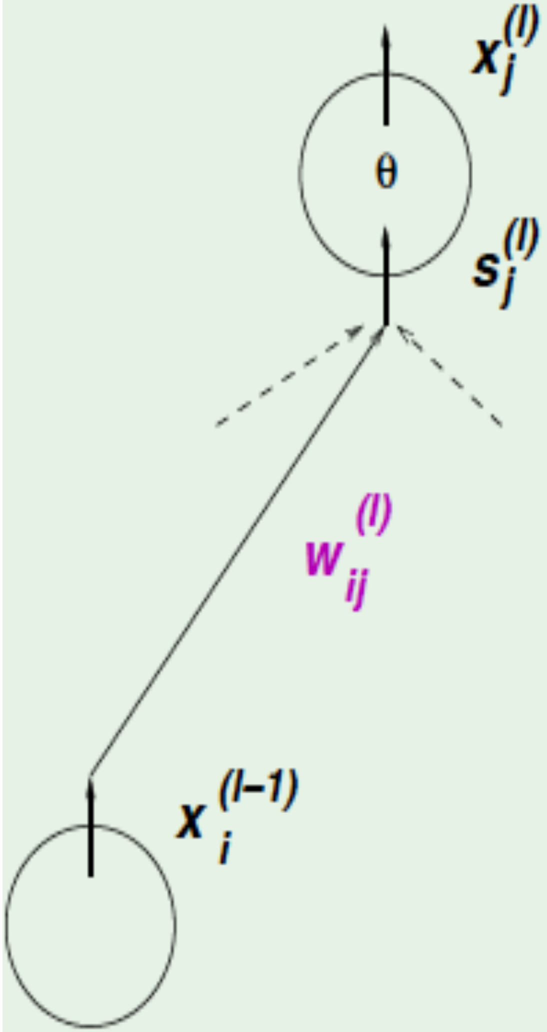


Computing $\dfrac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ w_{ij}^{(l)}}$

A trick for efficient computation:

$$\frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, w_{ij}^{(l)}}$$

We have $\dfrac{\partial\, s_j^{(l)}}{\partial\, w_{ij}^{(l)}} = x_i^{(l-1)}$ We only need: $\dfrac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} = \delta_j^{(l)}$



We will compute this recursively, starting from the last layer backwards

δ for the final (output) layer

$$\delta_j^{(l)} = \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_j^{(l)}}$$

For the final layer $l = L$ and $j = 1$:

$$\delta_1^{(L)} = \frac{\partial\ \mathrm{e}(\mathbf{w})}{\partial\ s_1^{(L)}}$$

$$\mathrm{e}(\mathbf{w}) = (\ x_1^{(L)} - y_n)^2$$

Assuming squared error function

$$x_1^{(L)} = \theta(s_1^{(L)})$$

$$\theta'(s) = 1 - \theta^2(s)$$ for the tanh

Back propagation of δ - Assuming all δ values of layer l have been computed already, how to compute δ for the i-th neuron (for any i) in layer (l-1)?

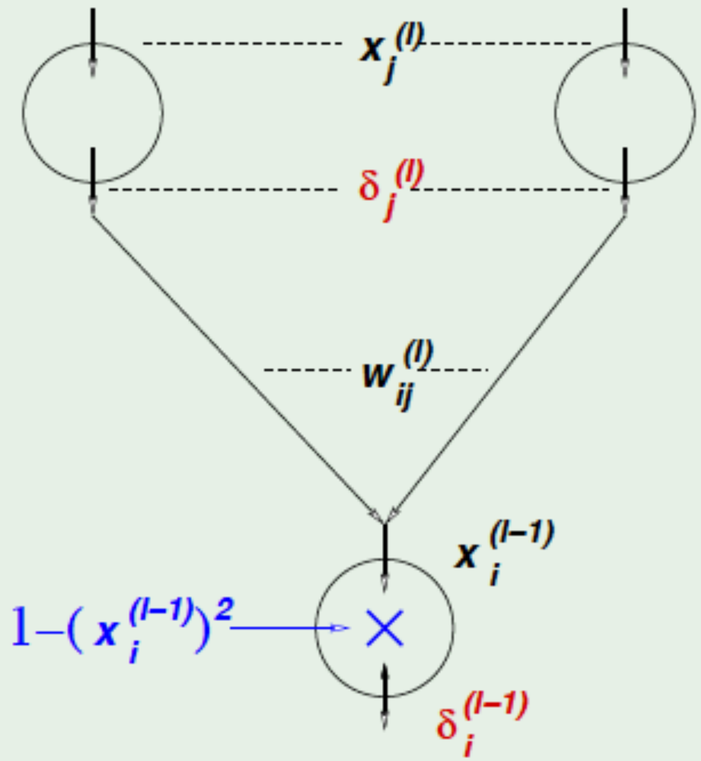
Back propagation of δ - Assuming all δ values of layer l have been computed already, how to compute δ for the i-th neuron (for any i) in layer (l-1)?

$$\delta_i^{(l-1)} = \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \frac{\partial\, \mathrm{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, x_i^{(l-1)}} \times \frac{\partial\, x_i^{(l-1)}}{\partial\, s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)})$$

$$\delta_i^{(l-1)} = (1 - (x_i^{(l-1)})^2) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)}\, \delta_j^{(l)}$$



Backpropagation

- δ values of layer (l-1) are computed based on the δ values of layer l
- So the δ values propagate backwards through the network

Backpropagation algorithm

Initialize all weights $w_{ij}^{(l)}$ at random

for $t = 0, 1, 2, \ldots$ **do**

  Pick $n \in \{1, 2, \cdots, N\}$

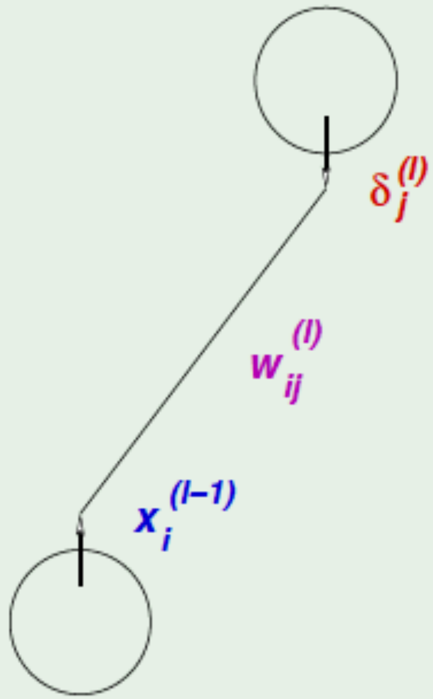
  Forward: Compute all $x_j^{(l)}$

  Backward: Compute all $\delta_j^{(l)}$

  Update the weights: $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\, x_i^{(l-1)} \delta_j^{(l)}$

  Iterate to the next step until it is time to stop

Return the final weights $w_{ij}^{(l)}$



Note: Each iteration uses only one training sample: SGD

Discussion

- Zero initialization will not work
  - If all weights initialized to zero, either all x's or all δ's will be zero; hence weights would not be adjusted
  - Weights have to be initialized randomly, or with some intelligent values (pre-trained models)
- How many layers? How many neurons in each layer?
  - Decide number of parameters (weights) based on available training data
- Not guaranteed to reach global minima; will reach a local minima depending on initialization, which sample chosen in which iteration, etc.

What are the hidden layers doing?
Learning non-linear transforms